The Å Publish/Subscribe Framework

Clint Heyer

Strategic R&D for Oil, Gas and Petrochemicals, ABB
Oslo, Norway
clint.heyer@no.abb.com

Abstract. This paper describes the design and implementation of a novel decentralized publish/subscribe framework. The primary goal of the design was for a high level of end-developer and user accessibility and simplicity. Furthermore, it was desired to have strong support for occasionally-connected clients and support for mobile and web-based systems. Content-based event patterns can be defined using scripts, with many common script languages supported. Script-based, stateful event patterns permit rich expressiveness, simplify client development and reduce network usage. The framework also offers event persistence, caching and publisher quenching. We also describe a number of applications already built on the framework, for example publishers to support location and presence awareness and ambient visualizations of financial data.

Keywords: Publish/subscribe, distributed applications, middleware, pervasive computing, ubiquitous computing

1 Introduction

Å was designed and developed as middleware to support rapid prototyping of pervasive computing applications for an industrial workplace. We desired a content-based publish/subscribe paradigm following on from our earlier experiences and success in using a preexisting publish/subscribe system for a similar purpose [1]. As publicly available publish/subscribe systems did not suit the requirements of the current research project, we designed and developed our own implementation.

Publish/subscribe systems are well suited for mobile usage [2]. Mobile devices tend to have unstable and variable network connectivity, for example a device might transition from a high-speed WiFi-equipped space to an outdoors area with lower-speed cellular data connectivity to a subway with no connectivity at all. Publish/subscribe systems generally hide connectivity issues, for example seamlessly rediscovering new, local routers and resubscribing, or publishing events that have been queued during a period of disconnection.

This paper is structured as follows. In section 2 we introduce the publish/subscribe paradigm for the unfamiliar reader. Section 3 outlines the requirements we devised for the system, followed by Section 4 with a description and discussion of the design and implementation of the Å framework. Section 5 describes some prototype applications we have already built on top of the framework, while Section 6 outlines plans for future work. Section 7 concludes the paper, summarizing the framework's features and relating them to the initial design requirements.

2 Publish/subscribe

Publish/subscribe systems allow *publishers* to send *events* (also referred to as *notifications*) to an event *router* (also known as a *dispatcher*) which is then responsible for distributing events to interested *subscribers*. *Subscribers* register their interest in particular events through the use of *event patterns*, which can be dynamically registered and deregistered. When an event is received, it is checked against registered event patterns and forwarded to the matching subscribers. Events usually consist of a collection of key-value pairs, with typed or opaque blob value types. One alternative approach is a tree-based structure for events, such as an XML-based format [3]. Traditionally the router was a single daemon [4], well-known to both publishers and subscribers but most contemporary systems use a distributed network of routers to improve reliability, scalability and performance [5; 6]. Publishers and subscribers are logical entities; in practice a single application can exhibit publisher or subscriber behavior, or some combination thereof.

The primary benefit of the publish/subscribe paradigm is the loose coupling of publishers and subscribers. Publishers produce events without prior knowledge of how the event will be distributed or to whom. Subscribers can consume events that they are interested in without prior knowledge of where the event came from or who produced it. Essentially, publishers send events into a cloud, and subscribers receive events from cloud. Communication takes place asynchronously with no requirement for coordination between publishers and subscribers, although coordination logic can be built on top of the eventing layer. Disruptions in communication are usually hidden, for example if a particular publisher is offline, subscribers will not necessarily be aware of it or affected by it – the absence of received events can be caused by any number of factors.

There is, however, a weak coupling in that subscribers usually need some prior knowledge of the format of events that the publisher produces. For example, a subscriber will not receive stock price notifications for symbol "XYZ" if it subscribes to the pattern `StockSymbol = "XYZ"` yet the publisher produces events such as `Symbol = "XYZ"; Price = 23.09`. Issues associated with event schema mismatches can be resolved by using *adapters*, which republish events with an alternative schema. This allows preexisting publishers and subscribers to be linked without modification or strengthened coupling. Publishing events with 'fuzzy' or higher-semantic values is also a potential approach. For example, a publisher sending events for person's current location might not only include latitude and longitude, but also the building, suburb, city and country or the fact that the person is in an office rather than outside. Subscribers interested in a person's location then have a variety of ways of expressing that interest, for example a region defined by latitude-longitude coordinates or perhaps they only care when someone is outside in a particular suburb.

Publish/subscribe event patterns are generally either subject-based, which uses broad groups or channels to determine event delivery, or content-based, whereby event patterns are used to filter events. The content-based approach permits subscribers to have fine-grained control over what messages they will receive based on the content of the events. Typically, this is accomplished using a simple logical expression that must evaluate to true based on event values. Expressions are, in effect, parsed and compiled and then executed against each incoming event to determine whether to deliver the message to the subscriber. Systems will often support basic functions which can perform value transformation and thus increase the expressive power of the pattern, such as accessing a particular sub-range of a string or converting a timestamp to elapsed time. Subscriptions are usually evaluated in a stateless environment, however some work has investigated stateful evaluation [7; 8].

For some kinds of publishers, the production of individual events may be a costly process so it would be beneficial to only publish events that matched current subscriber's event patterns. This process is called *quenching,* and allows the publisher to determine when there is some entity interested in its events, so that event production can be adapted accordingly.

3 System Requirements

We had a number of requirements for the system which were not satisfied by available publish/subscribe implementations. Our primary goal was for end-developer and user accessibility and simplicity. It needs to be as straightforward as possible to write applications that take advantage of the framework, applications should have minimal configuration requirements, be reliable and easy to deploy. Our secondary goal was to support loose-coupling, not only between applications built on the framework (as is usually the case with publish/subscribe systems), but also between applications and publish/subscribe system itself.

In addition to the two main goals, several other requirements shaped the design of the framework. Since the applications to be built on top of the framework were to emerge as part of an exploratory research process, it was critical that the framework was open and flexible enough to support a variety of uses. The Microsoft .NET framework is the predominate programming environment in our laboratory, so it was a requirement that the framework was accessible to .NET-based applications and followed API design conventions for that platform. However, as we intended on using the framework from a number of different environments, such as the web and embedded and mobile devices, it also needed to be programmatically interacted with it in a simple, open, standardized manner.

There were also a number of characteristics of many publish/subscribe implementations that we did not deem salient for our prototype applications. We did not for example, anticipate requiring any guarantees of ordered, reliable or timely event delivery, nor did the system need to be able to route messages with a high throughput or scale to millions of nodes. These relaxed requirements are contrary to the focus of numerous research efforts [9; 10; 11; 12; 13], however they suited our applications and allowed us to focus our implementation efforts appropriately.

4 Architecture and Implementation

This section describes the architecture and implementation of the framework and relates it to the literature. The distributed, decentralized network topology is described, in which clients connect to a router, which is a member of an interconnected mesh network. Events in the framework are typed key-value pairs. Each event has associated metadata which can be used in event patterns or extracted by subscribers, and is set by publishers and the router mesh. Subscribers can use a XML-based query language or scripts to define which events they receive. Script-based event patterns are a novel feature of the framework, and support a high level of expressiveness, reduced network traffic and simplified client implementation. Basic quenching support is provided so that publishers can be aware when a subscriber exists that is interested in their events, and perhaps adjust event production accordingly. Publishers can opt to have their events persisted in a mesh-based store which is transparent to subscribers. The mesh itself additionally caches events transparently to publishers, and clients can request their own mesh-based caching of particular events. These event storage and caching features permit occasionally-connected subscribers to still make use of events which occurred before they connected and removes the need for publishers to be continually connected. Client development is supported by a .NET API, tools and services. Mobile and non .NET-based usage is provided by a standards-based HTTP endpoint.

4.1 Peer-to-peer topology

We took a distributed, decentralized approach to event routing. Routers discover and connect to each other forming an interconnected peer-to-peer mesh, around which all events are distributed. Clients (publishers and subscribers) connect to a router within the mesh, forming leaf nodes (Fig. 1). If one router receives an event from a connected subscriber, it distributes it to all routers it has a connection to and so the process continues until the event has passed through all routers or its time-to-live hop counter has expired. Events are marked with globally-unique identifier when first received from a client, which is used to ignore duplicate forwarded events. Router nodes with multiple network interfaces automatically behave as network gateways, allowing individual Å entities to communicate transparently across network boundaries.

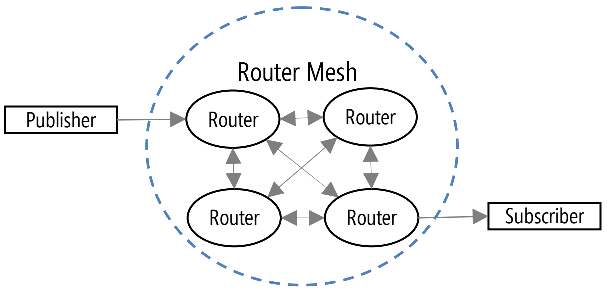

Fig. 1 Example network structure. Arrows indicate potential event flow.

Our peer-to-peer event forwarding approach is less sophisticated than other work, for example some implementations only forward events that the connected router has expressed an interest in [14], or use an optimized peer-to-peer connectivity topology [5]. As we did not expect to scale the system to thousands of routers or have a high throughput of events, we preferred the naive approach for its simplicity and reliability.

Routers maintain the mesh by continually looking for new router nodes to incorporate and exchanging addresses of known routers. Discovery takes place by broadcasting UDP packets on available or predefined network subnets and by attempting to connect to well-known router endpoints if they are defined. Mesh topology information is periodically persisted to disk so the router can quickly re-integrate into the mesh should it restart. When a connection to a router is lost, or an exception otherwise occurs during a communication, it is added to a “poison” list. Subsequent connection attempts occur in exponentially increasing intervals – thus a faulty node is increasingly marginalized the more frequently it fails, up until a maximum interval. The same discovery, persistence and poison list processes are followed in the client API used by publishers and subscribers. Once connected to a router (and thus, a leaf node of the mesh) clients periodically request alternative router endpoints which are attempted in case of disconnection to the discovered router. Together, these processes allow routers and clients to operate requiring zero or only minimal configuration.

Router nodes have three modes of execution: 1) a system service, managed by the operating system, 2) a console program, or 3) by referencing the router library, embedded in any .NET application. This flexibility allows nodes to be easily deployed and managed. In our environment, most nodes are run as system services, with scheduled tasks to automatically update router software.

4.2 Events

Events in the Å framework are essentially dictionaries, a collection of typed name-value pairs. Each event also has associated metadata, also a collection of name-value pairs. Publishers can use metadata to set per-event parameters which can be acted upon by routers or the receiving subscriber. On receiving an event from a client, the router will set metadata such as the client's unique identifier, whether the client has authenticated itself and so on. As events are forwarded around the mesh, metadata is used to track which routers the event has already passed through and also the time-to-live hop counter, which is decremented with each forward, both of which prevent infinite forwarding. Events can also have a *type identifier*, the value of which is treated as opaque by routers, but allows publishers to declare that particular events are of the same type, or group. For example, a publisher might send sensor readings every minute, and use event types to group events from a particular source sensor. Types can be used by the persistent store (described in section 4.5), used in event patterns, or extracted by subscribers when they receive an event.

4.3 Event patterns

Å uses a pluggable event pattern system to support alternative pattern evaluation kernels. Currently, there are two implemented kernels: script and XPath. The latter executes XPath [15] queries against events which have been transformed into an XML structure, however it adds additional overhead due to the XML transformation.

Script event patterns support a rich expressiveness and powerful filtering capabilities. Subscribers register patterns on a router simply by sending the text syntax of the script, its format, and the well-known language identifier. The pattern format determines if the script is evaluated as a simple logical expression or interpreted as a structured script which potentially could include functions and types. We leverage the Microsoft Dynamic Language Runtime (DLR), thus event patterns can be written in Python, Ruby, JavaScript or any other language with a DLR provider. Patterns have the full capability of the language they are written in, with all the in-built functions and supporting types available. Developers and users can therefore leverage existing knowledge of common scripting languages instead of learning and working within the confines of bespoke pattern syntax.

Rather than the typical approach of a pattern being able to only access individual event values, isolated from other events and the system as a whole, Å's novel script-based approach allows patterns to optionally incorporate a rich, broad awareness of the publish/subscribe environment. An object model is exposed so that the script can access and manipulate the event in a structured manner, maintain state, and access properties and functions of the script's host router.

With support for sophisticated pattern logic, subscribers can be simplified, reducing the need for additional client-side filtering logic. Moreover, executing logic "close" to the data and stateful subscriptions can reduce network traffic and offload computational work from the client. In turn, this enables subscribers to run in resource-impoverished environments like a mobile phone or embedded system. For example, a subscriber might start an alarm when a sensor reaches a high temperature. A traditional stateless event pattern, such as `Temperature > 100` would result in the subscriber receiving events continuously while the temperature is above 100. A stateful event pattern on the other hand could be used so that the subscriber only receives an event once the high temperature is reached, such as:

```
 if (Temperature > 100)
    if (!highAlarm) { highAlarm = true; ForwardEvent(e); }
else highAlarm = false;
```

Loosely-coupled coordination of distributed subscribers is also possible. For each router, only one instance of a particular pattern syntax is used. All subscribers who register the same pattern are associated with the single pattern instance, forming an ad-hoc group based on this shared interest. By utilizing the additional logic afforded by script-based patterns, the pattern might, for example, notify its subscriber pool in a round-robin fashion to distribute load.

4.4 Quenching

A basic form of quenching is supported so publishers can be notified when a subscriber registers for their events. Publishers can request to receive quench notifications by registering an event as an *event prototype* with the router. Routers in turn forward registrations around the mesh and ensure newly-joined routers are updated. On the initial quench notification registration, each router examines its list of subscriptions and sends a notification to the publisher for each subscription that matches the prototype. Any subsequent subscription within the mesh is checked against registered prototypes and generates the same notifications. The publisher is also notified if a matching subscription is removed or expires. Publishers are thus informed when there is an entity interested in their events and can alter publishing behavior accordingly, such as stopping or slowing. For example, an embedded device might measure snow depths in a remote valley, reporting readings over an expensive cellular data network. Reducing data transmissions would be beneficial, so the publisher can use quenching and only transmit events when there is an interested subscriber.

This approach is limited in that subscriber's event patterns need to be sufficiently broad in order to match the publisher's registered event prototype. Consider the aforementioned snow depth publisher. If a subscription has an event pattern of `SnowDepth == 80` and the publisher uses a quench event prototype of `SnowDepthSensor = True; SnowDepth = 50` the subscription would not match, the publisher would not receive the quench notification and thus events would never be sent. Alternatively, if the subscriber were to use an event pattern of `SnowDepthSensor == true`, the quench event prototype would match, resulting in the expected behavior. This, however, would require the subscriber to perform additional filtering if they are indeed only interested in events where the snow depth is 80 and increase the coupling between publisher and subscriber, as they now rely on the shared value of `SnowDepthSensor`. In our current usage of the framework, we consider this trade-off to be acceptable if quenching is required.

4.5 Event persistence and caching

Each router maintains a persistent event store, used to store events when requested by publishers. As events get forwarded around the mesh, they are replicated at each node, ensuring redundancy. Publishers can use event metadata to specify a combination of time or quantity-based store expiry policies. For example, a publisher could specify that the router should remove events after three hours and only keep a maximum of five events of that type. When a subscriber first registers an event pattern, the store is queried for matching historical events, which are sent to the subscriber as normal events (albeit with metadata identifying it as historical). The persistent store is particularly useful for publishers that run infrequently, send events infrequently or anticipate loss of connectivity which would preclude periodic sending.

Subscribers who might benefit from such a store (such as those that have intermittent mesh connectivity) need to rely on publishers to set the persistency option for events they send. This can introduce unnecessary coupling between publishers and subscribers if a publisher needs to be modified for the benefit of an occasionally-connected subscriber. Å has two techniques to address this problem: the mesh event cache, and the client event cache.

The mesh event cache functions like the event store, differing in four ways: 1) cached events are not transparently returned to subscribers, they must be specifically requested; 2) publishers have no control over expiration policy, only whether to opt-out entirely; 3) cached events are not persisted to disk, only stored in memory; and 4) it is opt-out rather than opt-in.

The client event cache allows clients to set their own caching policy, as they are not able to influence what is kept in the store or mesh cache. It is identical to the mesh event cache except that clients have ownership and control. This cache is maintained on per-client basis when requested, and replicated across the mesh so that it is still available even if a client reconnects to a different router node. Clients specify an event pattern and expiry policy, and are responsible for periodically sending keep-alive requests to prevent the cache being removed. Like the mesh event cache, events must be specifically requested – they are not returned as part of a normal subscription. Clients can also perform basic manipulations such as altering the caching policy and clearing it.

4.6 Security

Only basic security mechanisms are included in the current implementation of the framework. Events received by a router are marked with the publisher's network address so the event can be traced to a particular host if required. Publishers can optionally send a certificate after establishing a connection to the router. Routers inspect the certificate and if it was signed by the same root certificate authority as the router's own certificate, the publisher's events are marked with the username contained in the certificate. This identifier can be used in event patterns by subscribers, for example opting to only receive events from trusted sources. Routers can also be configured to only accept mesh or client connections from certain network address ranges.

4.7 Client API, services and tools

A .NET client API has been implemented to simplify building applications on the Å framework. It hides protocol details, provides useful data structures and performs management tasks such as mesh discovery. Using the API allows a simple publisher or subscriber to be written in as little three lines of code.

The framework includes a basic container service which can run as a console application or operating system service and hosts one or more add-ins. Developing Å components as add-ins simplifies their implementation and deployment as well as reducing system load. Deployment of add-ins is accomplished simply by copying them to the container service's directory and restarting the service. Event adapters and producers are well-suited as container service add-ins.

Command line tools are included so the framework can be interacted with directly and to support operating system scripting. Events can be sent or subscribed to, mesh metadata queried and simple per-event received actions can be performed, such as starting a program or appending to the system event log.

4.8 Mobility and HTTP access

To support mobile access and improve accessibility of the framework, each router node also exposes a XML-over-HTTP web service endpoint in addition to the binary TCP endpoints used for normal communication. HTTP communication and XML manipulation is supported by most modern programming environments and has a wide variety of supporting tools. Moreover, HTTP is permitted in many network environments which block traffic on non-standard ports. The endpoint uses a RESTful paradigm [16] well-suited for use not only from sophisticated applications but also command line scripts and web-based scripts and applets. HTTP-based access is included in the client API, which can also run from Windows Mobile-based mobile devices and simplifies development by hiding protocol specifics.

HTTP-orientated subscribers register event patterns in a similar fashion as binary-orientated subscribers. However, because of HTTP's stateless nature and the assumption that the client has only limited capabilities, it is not possible for the router to forward events as they are received and processed. Instead, for HTTP clients, the router implicitly enables the client event cache (discussed in section 4.5) so that events are stored on the mesh. Clients periodically query the HTTP endpoint which returns queued events in a standard Atom syndication format [17] XML stream. Mesh-side event caching is particularly useful for mobile clients as they can receive events even for periods that they did not have network access.

5 Applications

We have built a number of applications using the Å framework within our laboratory. All employees in our organization use notebook computers as their primary computing resource, and often operate in an occasionally-connected manner. The framework has proven to be reliable and robust in these situations, automatically discovering and connecting the mesh when it is available, and consuming little resources when unavailable.

Container service-hosted publisher add-ins run on a number of servers, workstations and notebook computers. Basic publishers have been written to produce events relating to financial data, current and forecast weather conditions, shared file storage usage, nearby WiFi or Bluetooth radios and instant messaging status. An application which runs on Windows Mobile devices sends similar network and location events, however also leverages in-built GPS receivers and cellular tower information. A wrapper around the Phidget[1] API allows a wide variety of hardware sensors (such as light, force, proximity, touch and so on) to produce Å events. Three industrial robots have also been augmented to produce events based on their current activity and status.

Adapters also run to "remix" events into alternative translations or add additional semantics. For example, a location adapter transforms radio proximity and network information events into location events, for example inferring that if a node is close to a particular WiFi access point, it must be in a certain building, and thus in a certain geographical location. Presence events are produced by a presence adapter, which asserts that if two separate nodes are detecting similar radios or have similar network properties, they are co-located or in close proximity. Because these adapters run on single server, matching logic can be easily altered without requiring updating individual nodes which produce the events.

Subscribers (mostly running on end-user notebook computers) make use of the events produced by publishers and adapters. A telepresence ambient visualization uses a LED bar to light a white wall with different colors based on activity occurring in the remote robot lab. This has been a useful mechanism for people working in the office space to be aware of their colleague's activity in the robot lab, located in a separate building. Another ambient visualization uses financial data events and Phidget servo-motors to articulate a sheet of satin. The sheet represents market index activity, while a LED bar behind the sheet colors it depending on the organization's stock price activity. The visualization thus provides focus, a company's stock price, and also the context, the wider market's activity. It has proven to be a popular in the office place, spurring conversation and raising awareness of the company's financial performance. Nabaztag/tag[2] rabbits can be controlled by sending events, for example moving ears, displaying light patterns or speaking aloud text. This is currently used for informal office place announcements. More sophisticated applications are currently under development, for example a location and presence awareness system which utilises location and presence events and sensors.

[1] http://www.phidgets.com
[2] http://www.nabaztag.com

Rapid development of prototypes was possible as the client API hides much of the implementation specifics, allowing them to operate without regard to connections, queuing, resources, authentication and so on. For a subscriber, data is simply put into a dictionary-like data structure and sent; for a publisher, after sending a subscription event pattern, data is received using standard callback mechanisms. Using the framework made network-enabling a data source such as a sensor straightforward, as well as implementing a remote client to make use of the data. Loose-coupling meant that there are few issues with versioning or coordination and has also allowed us to reappropriate event data without requiring modification of the publisher or existing subscribers. The container service speeded component development as scaffolding and lifecycle management code did not need to be written. The framework's open-ended event format allowed a variety of data to be exchanged and the sophisticated event pattern functionality allows subscribers exert fine-grained control over which events are received. Decentralized architecture eased configuration and increased reliability, as the system could still function even with unstable routers or clients. Having a unified event distribution mesh not only allows disparate components to communicate, but permits rich aggregation and translation of the disparate events, such as producing context inferences from numerous independent data sources.

6 Future Work

While the framework fulfils our current requirements, we are interested in advancing certain aspects of its functionality and conducting an evaluation and improvement of its performance.

Particularly of interest is to extend the script event pattern model to offer more sophisticated capabilities and better performance (rough benchmarking show that the script kernel can process several hundred events per second for a single basic expression). For example, scripts can be precompiled into intermediate bytecode to increase execution speed. If arbitrary bytecode can be used as subscriptions, this model could be further extended so that subscribers could send complete, precompiled modules for high-performance and rich mesh-side event filtering or to act as general purpose add-ins to the mesh. For example, an occasionally-connected client might inject a module to perform event adaption on the mesh itself. In-built .NET execution isolation sandboxes can also be leveraged so scripts run in restricted environments and unable to cause router instability. This would introduce significant additional complexity as remotely-loaded add-ins would need to be managed across the distributed mesh.

To further simply development from non .NET platforms, a JavaScript API could be developed to simplify access of the HTTP endpoint. Providing mechanisms for HTTP endpoint discovery is also a priority.

7 Conclusion

This paper described the design and implementation of the Å framework, a novel approach to the decentralized publish/subscribe paradigm which emphasises accessibility, simplicity and loose-coupling. The framework uses stateful, content-based event patterns, which can be defined as rich scripts, using a number of popular scripting languages. This permits a high level of expressiveness and allows clients to place more of their logic close to the data, improving performance and simplifying client implementation.

The primary design goal was for end-developer and user accessibility and simplicity. This is achieved through a powerful client API, automatic mesh discovery and maintenance, rich stateful event pattern expressiveness and event caching and persistency options. The secondary goal was to support loose-coupling between publishers and subscribers as well as framework components. Å provides additional support for loose-coupling through the use of caches, quenching and rich event patterns. Occasionally-connected clients such as mobile devices are well supported through the use of the aforementioned features. Å systems (clients or router nodes) are entirely self-hosting, and do not require any external services or infrastructure (apart from necessary basic network services) and are thus easily deployed and maintained. Tertiary design goals were for an open, flexible interface for non-.NET systems. This is provided using a HTTP-based endpoint which uses common web standards and protocols, and is thus accessible to most programming and network environments.

The paper also briefly describes a number of applications already built on the framework, such as presence and location publishers and ambient visualization systems and outlines plans for future work.

Acknowledgements. The authors wish to acknowledge the support of the Norwegian Research Council and the TAIL IO project for their continued funding and support for this research. The TAIL IO project is an international cooperative research project led by StatoilHydro and an R&D consortium consisting of ABB, IBM, Aker Solutions and SKF.

8 References

1. Sutton, P., Brereton B., Heyer, C. and MacColl, I. (2002). Ambient Interaction Framework: Software infrastructure for the rapid development of pervasive computing environments. *Proc. of the Asia Pacific Forum on Pervasive Computing* (pp 1-8). Australian Computer Society.
2. Muhl, G., Ulbrich, A., & Herman, K. (2004). Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Computing , 8* (3), 46-53.
3. Tian, F., Reinwald, B., Pirahesh, H., Mayr, T., & Myllymaki, J. (2004). Implementing a scalable XML publish/subscribe system using relational database

systems. *Proc. of the ACM SIGMOD international Conference on Management of Data* (pp. 479-490). ACM Press.

4. Segall, B., & Arnold, D. (1997). Elvin has left the building: A publish/subscribe notification service with quenching. *Proc. of AUUG97.*
5. Chand, R., & Felber, P. A. (2003). A Scalable Protocol for Content-Based Routing in Overlay Networks. *Proc. of the Int'l Symposium on Network Computing and Applications.* IEEE.
6. Carzaniga, A., Rosenblum, D. S., & Wolf, A. L. (2000). Achieving scalability and expressiveness in an Internet-scale event notification service. *Proc. of the ACM Symposium on Principles of Distributed Computing* (pp. 219-227). ACM Press.
7. Leung, H. K. (2002). Subject space: a state-persistent model for publish/subscribe systems. *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research.* Toronto, Canda: IBM Press.
8. Ionescu, M., & Marsic, I. (2004). Stateful publish-subscribe for mobile environments. *Proc. of the W'shop on Wireless Mobile Applications and Services on WLAN Hotspots* (pp. 21-28). ACM Press.
9. Pereira, J., Fabret, F., Jacobsen, H.-A., Llirbat, F., Preotiuc-Preito, R., Ross, K., et al. (2000). Publish/Subscribe on the Web at Extreme Speed. *Proc. of ACM SIGMOD Conf. on Management of Data* (pp. 627-630). ACM.
10. Triantafillou, P., & Economides, A. (2004). Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems. *Proc. of the I'tnl Conf. on Distributed Computing Systems (ICDCS'04)* (pp. 562-571). IEEE Press.
11. Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., & Shasha, D. (2001). Filtering algorithms and implementation for very fast publish/subscribe systems. *Proc. of the SIGMOD Int'l Conf. on Managment of Data* (pp. 115-126). ACM Press.
12. Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., & Chandra, T. D. (1999). Matching events in a content-based subscription system. *Proc. of the ACM Symposium on Principles of Distributed Computing* (pp. 53-61). ACM Press.
13. Eisenhauer, G., Schwan, K., & Bustamante, F. (2006). Publish-Subscribe for High-Performance Computing. *IEEE Internet Computing , 10* (1), 40-47.
14. Cugola, G., Di Nitto, E., & Fuggetta, A. (2001). The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering , 27*, 827-850.
15. Clark, J., & DeRose, S. (1999). *XML Path Language (XPath).* W3C.
16. Richardson, L., & Ruby, S. (2007). *RESTful Web Services.* O'Reilly.
17. Nottingham, M., & Sayre, R. (2005). *RFC 4287 Atom Syndication Format.* IETF Network Working Group.